An approach for Supporting *OpenMP* on the Intel SCC

Hayder Al-Khalissi
Chair of Chip-Design for Embedded Computing
MÃijhlenpfordtstraÃ§e 23
38106 Braunschweig
alkhalissi@c3e.cs.tu-bs.de

Andrea Marongiu
DEIS-University of Bologna
Viale Risorgimento 2
40133 Bologna
amarongiu@deis.unibo.it

Mladen Berekovic
Chair of Chip-Design for Embedded Computing
MÃijhlenpfordtstraÃ§e 23
38106 Braunschweig
berekovic@c3e.cs.tu-bs.de

ABSTRACT

The advent of the Single-chip Cloud Computer (SCC) chip in the many-core realm imposes challenges to programmers. From a programmer's perspective is desirable to use the shared memory paradigm, employing high-level parallel programming abstractions such as OpenMP. In this paper we discuss our ongoing efforts to support OpenMP on SCC. Specifically, we focus on the following three key aspects in our approach: i) Investigating an implementation that is aware of the memory hierarchy. ii) How to handle OpenMP shared variables. iii) Efficiently implementing synchronization (i.e., barrier) constructs by leveraging SCC hardware support. To meet this need, we propose effective barrier synchronization implementations for OpenMP on the SCC. In particular, we present an efficient evaluation of the overhead associated with integrating barrier algorithms that is required for OpenMP run-time libraries on such a machine. Our initial experimental results show significant performance improvement up to 98% for 48 cores.

Keywords

MPSoC, OpenMP, Barrier synchronization.

1. INTRODUCTION

Intel's SCC platform [10] is dedicated to exploring the future of many-core computing. It is a research architecture resembling a small cluster or "cloud" of computers, therefore, it interesting in a variety of different application through HPC space. The SCC architecture has 48 independent Pentium P54C cores, each with 16kB data and program caches and 256kB L2 cache. The cores are organized as 24 dual-core tiles connected via a low-latency mesh network. Each tile connects to a router and contains two cores, a *Mesh Interface Unit* (MIU), and a pair of *test-and-set* registers for realizing atomic access. Moreover, SCC features four on-chip DDR3 *memory controllers*, which are connected to the 2D-mesh as well. Each controller supports up to 16GB DDR3 memory, resulting in a total system capacity of 64GB. Being based on the P54C architecture, each core is able to access only 4GB of memory. To solve this limitation, each core has *Lookup Tables* (LUT) with 256 entries with 16 MB granularity translate the address mapping 32-bit physical core addresses to the 64GB system memory. It is part of the configuration register space that is itself mapped by a LUT entry and shareable between cores. Each entry in LUT is configurable and points to specific types of memory spaces (off/on-chip memory, configuration and synchronization registers). The SCC does not offer cache coherency between the cores, but rather employs special 16kB-sized *Message Passing Buffer* (MPB) for improved communication efficiency between cores. A new `CL1INVMB` instruction together with a dedicated *message passing buffer type* (MPBT) are used to provide coherency guarantee between caches and MPBs. MPBT data is not cached in the L2 cache, but only in the L1 cache. Hence, when reading the MPBs, a core needs to clear the L1 cache. As the SCC cores only support a single outstanding write request, a *Write Combine Buffer (WCB)* is used in MPBT mode to combine adjacent writes up to a whole cache line which can then be written to the memory at once. When a core wants to update a data item in the MPB, it can invalidate the cached copy using the `CL1INVMB` instruction. Given this hardware configuration, the SCC is designed to support the message-passing based programming models. One well-known customized library providing the message-passing model is *RCCE* [30].

OpenMP [6] is a *de-facto* standard for shared memory programming, since it provides very simple means to expose parallelism in a standard C (or C++, or Fortran) application, based on code annotations (compiler directives). This appealing ease of use has recently led to the flourishing of a number of OpenMP implementations for embedded *Multi-processor systems-on-chips* (MPSoCs) [18, 20, 24, 22]. MPSoCs typically feature complex memory systems, with explicitly managed SRAM banks and NUMA organization. SCC is no different in this respect, and poses several challenges to accommodating the OpenMP execution model. First, each core runs a separate instance of the operating system, which makes it impossible to run existing OpenMP implementations based on standard threading library (e.g., Pthreads) directly. Second, barrier primitives should leverage fast and local memories such as the MPB to minimize inter-thread synchronization time. Third, data sharing is not at all trivial, as OpenMP assumes a flat memory model, which is unmatched by the distinct private virtual memory segments seen by different SCC cores.

In this paper, we will go over the design and an efficient implementation of the OpenMP execution and memory model for SCC, describing our initial experience with the GCC compiler and a custom implementation of the run-time library. Specifically, we first study the parallel code generation for OpenMP by GNU GCC. We then describe the design of our SCC OpenMP run-time library by coping with three challenges in our systems: i) Supporting unmodified legacy OpenMP programs on SCC. ii) Implementing the OpenMP memory model. iii) Reducing the overhead for synchronization directives.

We implemented several barrier variants for integration into the OpenMP run-time library. We evaluate also the performance achieved by different approaches and show the benefits and drawbacks of individual approaches as well as significant performance improvements for the optimal solutions. The rest of the paper is organized as follows: Section 2 gives an overview of the compiler infrastructure upon which our work is based. Next, we focus in more depth at each of the challenges of retargeting OpenMP model to the features of the SCC system in Section 3. Methodology and micro-benchmark implementations to evaluate the barriers performance are discussed in Section 4, while the Section 5 presents some experimental results to analysis the barrier synchronization in our approach. Finally, our conclusion and future works are given in Section 6.

2. OPENMP TRANSLATION

One of the techniques used to execute OpenMP applications in MPSoC environment is to analyse the accesses to the shared data and transform them into some other communication primitive supported by the native runtime system of the platform. At compile time, the OpenMP program can be translated into other languages suited for target platforms, MPI or global arrays (*GA*) for example. Basumallik [5] and Millot [23] transformed OpenMP to MPI library. Wang [31] and Dorte [13] implemented *LLCoMP* to translate extended OpenMP to MPI by using skeleton method . This kind of transformations are feasible when the application makes regular accesses, but when the accesses are irregular then the transformation becomes trickier. Huang [17] and Chapman [14] (based on OpenUH compiler) made the same transformation, but instead of using MPI, they used global arrays. There is another way to implementing OpenMP by using *Source-to-Source* tool which takes as input C/C++/Fortran source code with OpenMP directives and outputs equivalent multi-threaded code, ready to be built and executed on a multiprocessor [4, 7, 27]. The generated code is compiled by native back-end compiler (i.e. GCC) linked with the runtime library.

In our approach, code transformation and calls to the runtime library are automatically instantiated by a customized GCC 4.6 compiler. GCC was chosen as a starting point as it provides a robust, open source implementation of the OpenMP translation pass and runtime library (**libGOMP** [15]). Figure 1 explains the transformation process of the compiler using the sample code. Using the `-fopenmp` flag enables OpenMP translation, and dynamically links the transformed program to the *libGOMP* library. Figure 1 shows how the compiler transforms the annotated application code. `#pragma omp parallel` blocks are outlined into new functions containing the code to be executed by parallel threads. The compiler encapsulate all shared data into a C-like `type-`

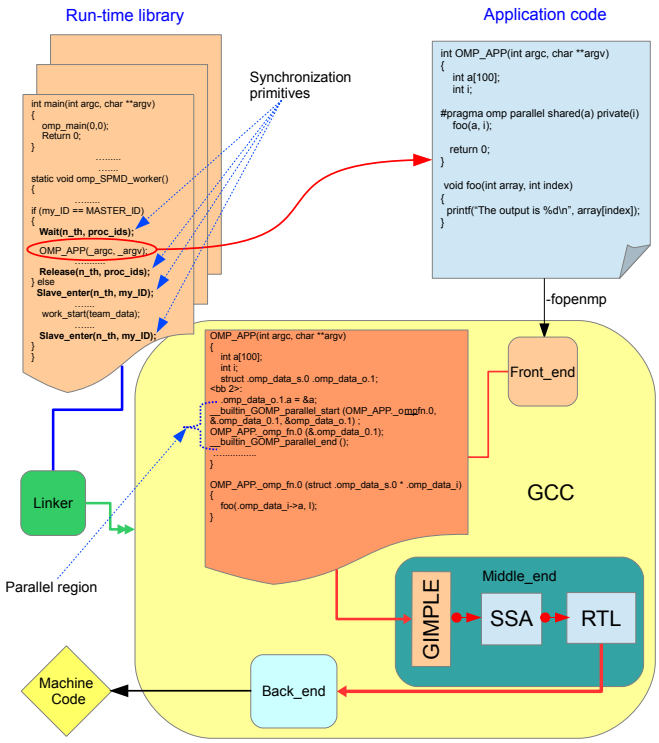


Figure 1: OpenMP code example and GCC compiler transformations

`def struct` and inserts a call to the runtime library function (`GOMP_parallel_start`), passing the new function and the struct as parameters. This enables new threads to execute the parallel function and to point to the shared data items. To close the execution of a parallel region from the master thread the compiler inserts a call to the `GOMP_parallel_end` function. Inside this primitive a barrier synchronization step is enforced. In original design of *libGOMP*, POSIX thread (Pthreads) are used as a standardized API for creation and manipulation of threads with certain operating systems (e.g. SMP GNU/Linux). Using *Pthreads* on SCC would require dedicated abstraction layers to allow the communication between threads on different cores. Also, the overheads associated with library (such as conditional variables and signal handling techniques) and as context switching that takes hundreds of cycles to execute [16].

To prevent these inefficiencies from limiting the parallelization effectiveness, we designed the new run-time environment (**libgomp_scc**) from scratch so as to efficiently deploy the parallelism of OpenMP applications on the SCC platform. *libgomp_scc* is a low-level library-based API that has capability to manage resources in SCC system. It stores all of the run-time data structures and it is used to create and manage the OpenMP threads. In a traditional OpenMP implementation, the master core is responsible for creating parallel worker threads when encountering a parallel region and for tearing them down upon the end of parallel region. Dynamically creating and destroying threads every time that a parallel construct is encountered is very costly, so we opt for a different solution. In our implementation of the *libgomp_scc*, we use a custom micro-kernel code executed by every core at start-up [9, 21] by assuming a fixed allocation of the master and slave threads to the processors. As a re-

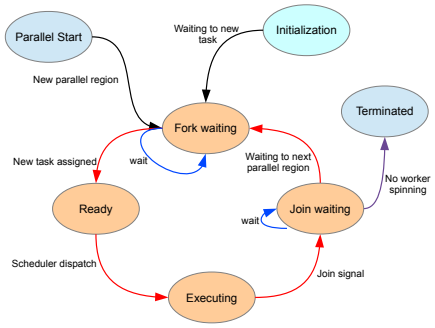


Figure 2: State transition diagram of OpenMP model on SCC

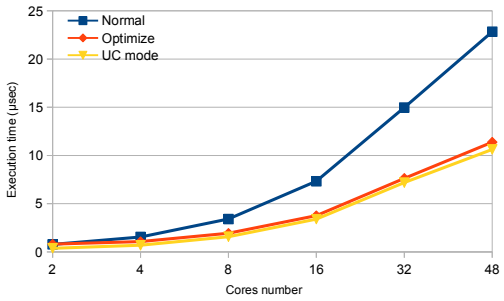


Figure 3: Impact of the cache alignment on MPB access

sult, threads can be quickly re-started at a later time when threads are docked upon parallel region end. Specifically, we create persistent threads at program launch by loading the executable image of the program + library onto each processor local L2 memory at boot time.

3. OPENMP IMPLEMENTATION

The cores of SCC are single-threaded. Therefore, in the rest of the paper we consider cores and threads to be equivalent, as we do not "oversubscribe" cores but only assign one thread per core. As illustrated in the previous section, the GCC compiler translates an OpenMP directive into multi-threaded code containing function calls a customized run-time library. Then, the run-time needs to map the OpenMP parallelism onto the SCC architecture. To do so, we hide interaction with the OS in the implementation of the runtime, thus abstracting architectural concerns from the programmer's view. In the remainder of this section, we explain the needed modifications to the compiler and run-time design.

3.1 Parallelism model

OpenMP [6] employs the *fork/join* execution model that is easy and flexible to handle sequential and parallel parts of an application. The program executes sequentially within a single thread, referred to as the *Master thread*, until it encounters a `#pragma omp parallel` directive. Here, execution forks into a multitude of threads by assigning (forking) computation to a number of worker threads (*slaves*) after initializations have been done. As a result, a parallel region is created. At the end of the parallel construct the master waits for all slave threads to complete (join) before continuing execution. Then only the Master thread resumes execution. Traditionally, conditional variables and signal handling techniques are used in the original implementation of OpenMP. The POSIX thread library provides conditional variables that require a thread be waiting for the conditional variable to receive it. Meanwhile, the core cycles are released and can be scheduled for other task. One of the main drawback of this schema is the large context-switch overheads between the sleep and wake-up states.

To work around this problem we rely on a simpler *busy_waiting* mechanism. Figure 2 show the state transition diagram for OpenMP implementation on the SCC. As shown in the figure, there are a total of seven states. In the *initialization* state each processor loads its executable image (the program + library) onto its own local L2 cache memory. Master and slaves threads execute different code base on the hosting core IDs. The master jumps to the execution of the main program, while the slaves wait for notice from the Master thread about available parallel work. When the program executing on the Master thread encounters a parallel region, the master is set to the *fork waiting* state, where it recruits slaves for parallel execution (as many as the user has specified). After that the state changes to *ready*. Here slaves are pointed to the parallel function and to the shared data. This triggers execution on the slaves, and the master itself transitions to the *executing* state. When the execution is complete, slaves enter the *join waiting* state, where they busy-wait for a new parallel region or for the master to notify program termination.

Note that we use distinct memory locations for different threads to busy wait (polling). Each thread is thus only waiting for its own task rather than competing for global tasks. As a consequence, the system does not need to switch between sleep and wake-up stages every time, thus reducing the overhead associated to context switching. To implement the spin lock, we use small storage cells into local message passing buffers, thus minimizing the cost for busy waiting and the contention for the system interconnect.

3.2 Thread Creation and Management

In a traditional OpenMP implementation, *libGOMP* manages a pool of threads. Namely, we can add new threads only when the thread pool is empty and the number of threads is usually much larger than number of cores on the platform. A thread is not removed, but added automatically to a thread pool for later reuse at the end of a parallel region. However, this implementation of thread pool has limitations: all threads consume system resources (e.g. stack space), the utilization of the thread pool must be managed efficiently to avoid increasing number of idle threads which affect the runtime performance significantly. Moreover, P54C core on SCC is not support multi thread mechanism and it consequently run one task at time.

To handle this situation, we use elastic **metadata** primitive to query all of thread-local data associated with thread team on the platform. By this way, we guarantee that no system resources are wasted and entirely eliminate many of the management problems inherent in the traditional OpenMP implementation. In addition, this approach ensures that the spinning task executed by Slave thread (while not into parallel region) does not interfere with execution of sequential parts of the program on the Master thread. *Metadata* are resided on the local memory (MPB) of Master thread to avoid the extra overhead of *aligning the data allocation in*

MPB with cache lines as depicted in Figure 3. Figure 3 shows the results of influence the cache alignment on access to MPB in which an increasing number of cores. We implemented this micro-benchmark for read and write concurrently from/to MPBs (under the same setting in the Section 5). In this micro-benchmark, we measure the overhead (μsec) of : (i) one loop (Normal) is responsible to read one byte allocated on the MPB (MPBT mode) at single core and later write one byte to same address in each iteration, (ii) two loops (Optimize) are used to read and write separately to same allocation in the same MPB. As explained in the figure, *Optimize* implementation has overhead less than 50% (for 48 threads) compared to *Normal* implementation in MPBT mode. While, the impact of separating loop to two has added little bit overhead (resulting of adding extra loop) when the number of cores up to 2, Otherwise, we did not find such evident optimization impact in the performance of the same *Normal* implementation executed in *un-cached* mode (*UC mode*). Hence, every access in each loop will fall on the same cache set. Therefore, we allocate the *metadata* in the Master's MPB by using MPBT mode. Once the parallel region starts, the Master thread updates the *metadatas* which containing task and frame pointers in its MPB of all slaves. To address the issues of interference traffic on the interconnect and atomic access, we use multi copy of *metadata* and allocating them onto the MPB of Master thread. Because of the Master thread has intensively access to them. All slave threads can join the parallel region by access their *metadata* based on the core id. Moreover, this approach relaxes the condition of identical threads implementation and allowing each thread to be distinct (heterogeneous threads) relay on their core ids. It may also support nested parallelism as well by porting special *metadata* that contains multiple threads.

3.3 Synchronization primitives

The OpenMP has a number of means to perform synchronization between parallel threads, such as *critical*, *atomic*, and *barrier* [6] . These directives have been implemented in the original OpenMP run-time library based on a two-level lock: mutex and spin_lock, where the POSIX thread library provides the first one. However, to handle *critical* and *atomic* functions, we can use the synchronization hardware available (i.e, *test-and-set* registers) in the SCC. *Barriers* – implicit or explicit – are central constructs to the OpenMP execution model and to any shared memory parallel program. Implicit barriers are used at the end of parallel regions in order to ensure that the slave thread does not start until all application's threads have completed the first parallel block. While explicit barriers (`#pragma omp barrier`) are used by program developer in order to ensure all threads pass this synchronization point at the same time, even if arriving at different times e.g. due to different workloads. The *fork/join* model imposes two synchronization events per parallel loop as mentioned before. Consequently, the costs of barrier for *fork/join* model deserves more attention, especially when the application has nested loops such as parallel inner and sequential outer loops. Barrier synchronization overhead has been recognized as an important source of the performance degradation in the execution of parallel programs [26, 12, 28].

Several implementations of OpenMP for MPSoCs have adopted a centralized shared barrier [18, 20]. It relies on shared entry and exit counters, which are atomically updated through lock-protected write operations. The counters are used to hold the number of threads that have reached a barrier. The threads wait for the last thread reaching the barrier, subsequently signalling all other threads of its arrival by setting the flag. This algorithms yields bad performance as the access to the counter is *serialized* [9].

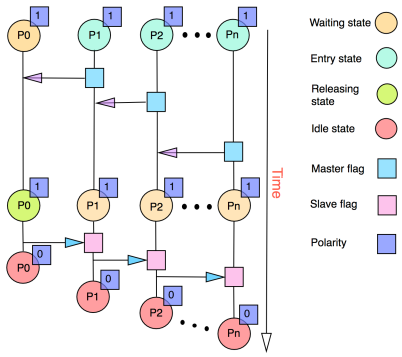


Figure 4: Chain-Polarity Barrier

However, in non-cache-coherent systems such as SCC, the updates to barrier structures (e.g., control flags, counters) in shared memory must explicitly be kept consistent. To work around this problem, we considered several different barrier algorithms which leverage specific hardware support for synchronization, pattern of communication, or its explicitly-managed portion of the memory hierarchy, as illustrated in [2] and [1] (submitted for publication).

We exploit a *Master/Slave* scheme to implement several algorithms on the SCC. This approach is accomplished in two phases, the *Entry* or *Gather* phase and the *Signal* or *Release* phase. As explained in Figure 1, each Slave signals its entry into the barrier by using a `Slave_Enter()` function in the entry phase; it then waits for the release signal. Gather and release phases are controlled by the Master thread through the corresponding `Wait()` and `Release()` functions (as shown in Figure 1). These functions have been implemented individually for being able to execute additional housekeeping functions before releasing the slaves. In this section, we give a brief description about an efficient implementation of barrier algorithms comparative to our previous work, therefore, more details can be found [2, 1].

3.3.1 Chain-Polarity Algorithms (CPB)

This is new approach that employs the polarity exit technique for implementing the Signal phase [3], that allows us to use a single flag array instead using of double arrays in the previous algorithms (S-MSB in [2]). The polarity approach is employed for handling reinitialization. We use a private boolean variable for indicating the current barrier state polarity in order to guarantee correct initialization for derived implementations. Every thread will validate their polarity with a shared release variable. To avoid a possible issue for the case that one thread enters a barrier with a polarity that is opposite to the polarity of another thread, we start the barrier using a default initial polarity value. Thus, all threads enter barrier with same polarity and this issue does not occur. Furthermore, the Master thread has no information about the slaves received the release signal. The Master only knows that all threads have received previous release signal once a thread enters the barrier again. As shown in Figure 4 (where threads are represented by a circle,

time flows downward, and memory allocation is symbolized by a square shape), **CPB** allocates each of the slave's poll flag onto their local memory and uses the chain mechanism to broadcast the release signal and gather the entry signals. We use a chain scheme to reduce the overhead of publishing the release signal in the Signal phase and avoid contention access in a single shared variable approach. In the chain mechanism, each thread receives the entry signal from its next higher-numbered neighbor, and then send its entry signal to the next lower-numbered neighbor. With each core having one predecessor and one successor, this effectively creates a chain topology.

3.3.2 Chain LUT Polarity Barrier Algorithm (C-LUTB)

In the SCC, every core has a *lookup table* (LUT) with 256 entries used for physical-to-physical address mapping [10]. The LUT is a shareable between all the cores and mostly used by operating system, but may be used also on application level. It is possible to change the contents of the LUT dynamically without causing problems to the core's memory management. This raises the idea of using the LUT entries for barrier synchronization. This is granted: not only can every core access its own or any other core's LUT, but also LUT entries are mapped by using `mmap()` in uncacheable to L1 and L2 caches. As a result, by using LUTs we avoid the issues of ensuring a consistent view of MPB. Moreover, the LUTs are located very close to the core and, thus, we avoid the extra overhead of access off-die registers and local memories. **C-LUTB** is a new implementation that is using a single LUT entry in every core to track notify and release signals. To implement this algorithm we need to use one entry in each core. We use the same gathering mechanism for Entry phase as in the CPB algorithms: instead of using local memory for allocating the flags, we used the LUT entries with a private variable (similar to exit polarity mechanism) to handle the reinitialization problem. In the Signal phase, the master thread waits to receive the notification signal from its higher neighbor and exchanges the LUT entry of the participant's last thread. Because the LUT entry of the last thread is not used (no further neighbor in the chain), we use this entry for releasing the slaves, to terminating all slaves. All slaves wait for exchanging the state of the last thread's LUT entry that indicates a release signal.

3.4 Memory model

OpenMP supports a relaxed memory consistency model similar to the weak ordering memory model. This model allows each thread to have a *temporary view* of the memory that it can use to store data temporarily and not visible to other threads. Writes to memory are allowed to overlap other computation and reads from memory are allowed to be satisfied from a local copy of memory under some circumstances, until it is forced to memory by OpenMP *flush* operation. Obviously, such a memory model can efficiently implemented on SCC memory structure, because the SCC has local memories which are usually addressable with different address space and can not be accessed by other core directly [19].

In OpenMP, there are two main data qualifiers: *shared* or *private*. A shared variable is accessible by all the threads inside a parallel block while the private data consists of distinct instances (one per thread) of a same variable. The OpenMP memory model assumes a single, flat shared memory space. On SMP machines multi-level coherent caches are used to reduce the average memory access cost while preserving the abstraction of a unique memory space. The SCC has no cache coherence between processors, alternatively it offers an on-chip memory to reduce the latency of memory access. We are currently working on efficiently supporting data sharing on SCC. The main difficulties that such a design poses are summarized in the following two subsections.

3.4.1 Local Memory

Listing 1 illustrates different ways of specifying shared data semantics in a OpenMP program.

Listing 1: Illustration of variables visibility.

```
int a;
void foo(){
  int b, c, d;
  #pragma omp parallel shared(b,d) private(c)
        a = b + c + d;
}
```

Global variables, like **a** in the example, are considered to be globally visible to every thread in a SMP system. For this reason, the original GCC implementation assumes that referencing the variable by name within any parallel thread is sufficient. It is not even required that the variable is declared as `shared` with the `parallel` construct. On SCC, on the contrary, the whole address space is aliased over different processors. As a result, referencing a variable at a given address from two distinct cores results in accessing different physical memory locations.

Non-global variables, declared within the scope of the subroutine which contains the `parallel`directives, get mapped on the master thread's stack. Even if in a SMP system different threads can access each other's stack, it is necessary that the code is transformed to pass the variable by reference. Note that on SCC this is not sufficient, because the stack of every processor is allocated on core-local memories, which in turn are accessible via an aliased range of the global memory map (i.e., the same address on different cores points to different physical addresses). This is the case of variables b and d in the example.

Variables declared as `private` in OpenMP imply that parallel thread owns a private replica of that object. The GCC compiler implements this by replicating the variable declaration within each parallel thread. We do not need to modify this behavior for SCC, since private data gets by default allocated onto local memories to each core.

We can modify the GCC implementation to treat global variables in a similar manner to what is done for automatic data declared as `shared`. However, we still have the problem of sharing pointers between distinct threads.

There are several ways to handle this issue. One is to use complex and CPU-specific handling of stack frames [7]. This scheme has two disadvantages. First, the produced code would no longer be similar for the two execution models. Second, it is clearly slow.

Another option [21], is that of augmenting the original GCC mechanism of marshalling shared data within a structured construct to always pass shared objects by reference. This ultimately allows to overcome the issues related to memory aliasing when data is referenced by name. However, for this approach to work, it is necessary to allocate program data in a portion of the physical SCC shared memory that

Figure 5: Overhead model for barrier implementation

can be ultimately made univocally addressable by different threads. The latter is the solution which we are currently exploring. The main drawback is currently the non-coherent nature of the transactions involving the shared memory that we discuss in the next section.

3.4.2 Non-Coherent Cache

Physical shared memory is available in the SCC, but it has not cache coherence. Therefore, the latency to access to shared variable should be a very high compared to accessing to private memory. Fortunately, the memory model of OpenMP requires a coherent view for shared variable only at specific synchronization points due to relaxed consistency memory model. Thus, is possible to manage the cache coherence by implementing software flush instruction in runtime library [29]. Another approach would be to rely on a software-managed cache. Allocating data on the on-chip local memory (MPB) could also be explored, but the reduced size and the necessity for explicit data transfers makes it a less appealing solution. Alternatively, reducing the shared memory access by put some data into the on-chip memory (MPB) through extending run-time library.

We are currently exploring the possible alternatives to achieve efficient data sharing on SCC. One promising solution is to rely on the main shared memory, possibly extending the programming interface with custom directives to originally place shared data items in a program in this memory region. The practical limitation is that this shared memory has little physical space, so we are also exploring the possibility of using part of the cores' private memory, by leveraging LUT to resolve virtual addresses and modifying the way the compiler passes shared data pointers in such a way that offsets are passed instead. This is fully ongoing work, so we don't provide further details here.

4. METHODOLOGY

Culler et al. [11] proposed LogP-model that accurately predict performance of a large and complex program on active-message based systems. We use this model to represent the overhead of barrier algorithms as depicted in Figure 5. The parameters of this model are listed in the Tabel 1. In most previous studies a simple micro-benchmark used to measure the time only in the Master thread. They measured the latency of gathering and releasing the participating processors as illustrated in the Figure 5 (red bar), but they overlooked the overhead by slave threads in the phases of the barrier in their works. However, we can calculate overhead

Table 1: The parameters of the barrier performance model

Par.	Description
O_g	Core overhead to read the flags in the Master.
L_g	Communication time to gather flags in the Master.
O_c	Core overhead to check the status of flags.
O_s	Core overhead to update the flags in the Master.
L_s	Communication time to update flags in the Master.
O_e	Core overhead to update the flags in the Slave.
L_e	Communication time to update flags in the Slave.
L_r	Communication time to read flags in the Slave.
O_r	Core overhead to get the new flag signal in the Slave.

of LogP-model parameters (Tabel 1) by measuring the time in two groups which are *Master Overhead* (MO) and *Average Slave Overhead* (ASO). The MO represents the cost for different approaches of performing barrier synchronization in the Master thread, including the two barrier phases. The influence of the overhead of inserting the new value of flag (yellow bar in the Figure 5) that is included in ASO sight. The ASO collects the overhead per participant (excluding the Master) by summing up all slaves' overhead divided by their number. Best to our acknowledge, we first use this procedure tat allows a direct comparison of barrier costs on Master and Slave side and therefore determining the quality of the evaluated algorithms with regard to execution time and parallel speed-up.

EPCC [8] is a simple micro-benchmark used for measuring barrier performance; however, as it does not regard the implicit barriers in OpenMP directives, we consider it insufficient for our cause. In this work, we implemented micro-benchmark based on the above scenario: *Pure Overhead*. Also, we extending this micro-benchmark to analyze the *Impact of memory access mode* on the performance of barrier algorithms. In the Pure Overhead micro-benchmark, barrier code is executed only on the platform without any communication between cores takes place. This allows estimating how the algorithm scales with increasing synchronization traffic only [2]. Moreover, we have already studied influences of different conditions on the overhead of barrier in [1]. For computing the average pure overhead of barrier, we use the following equations:

$$Avg_{MO} = \frac{TotalBarrierTime}{(No.ofIterations - No.ofIgnores)} \quad (1)$$

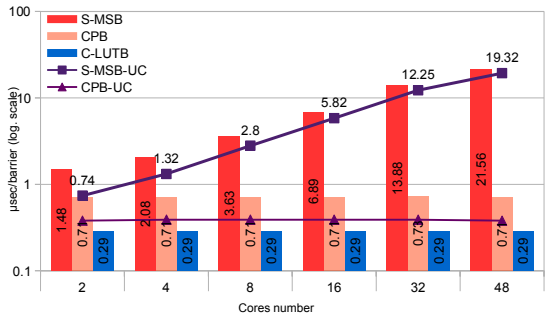
$$Avg_{SO} = \sum_{i \neq Master}^{No.ofT_{id}} (AvgBarrierTime) \quad (2)$$

$$O_P(\mu s) = \begin{cases} Avg_{MO}, & \text{if } T_{id} \text{is Master} \\ \frac{Avg_{SO}}{(No.ofT_{id}s)-1}, & \text{else} \end{cases} \quad (3)$$

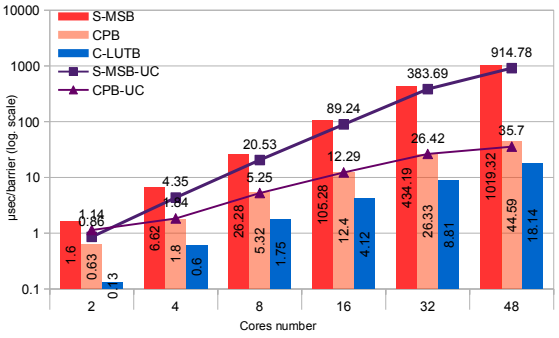
Where:

1. Avg_{MO}: is the average time of Master thread including `Wait()` and `Release()` functions.
2. Avg_{SO}: is the average time of Slave threads including only the `Slave_Enter()` function.
3. $O_P(\mu s)$: is the average execution time of Master or Slaves in μ seconds.
4. $No.ofT_{id}s$: is the number of running threads (participants)

The SCC features two types of accesses to its MPB, *cached*



(a) Master Overhead



(b) Average Slaves Overhead

Figure 6: Pure Overhead and Impact of UC-mode of Barrier algorithms

(MPBT) and *un-cached* (UC). In MPBT mode, data caches in the (L1) cache only and issues write ordering after filling the WCB. While in UC mode, data reads are not cached and writes are directly issued to the network. The default implementation of barrier algorithms which have access to MPB memory, it is based on MPBT mode. As explained in our previous work [2], the UC mode has impact in the overhead reduction more than 41%. We measure the influence of memory mode access on the performance of Pure Overhead micro-benchmark for CPB algorithm that implemented based on MPB.

5. PERFORMANCE EVALUATION

The experimental results are generated using the default SCC settings, which are standard LUT entries, 533 MHz tile frequency, 800 MHz mesh and DRAM frequency for all micro-benchmarks except **Impact of Frequency scaling**, where the influence of the different frequencies is regarded separately. The experiments are conducted using sccKit 1.4.2.2 running a custom version of sccLinux based on the 2.6.32.24-generic kernel. For timing analysis, *RDTSC* (Read Time Stamp Counter) instructions [25] are inserted before and after the barrier algorithm. We execute each barrier algorithm 100,000 times, determining the mean execution time in μs. We used S-MSB as a barrier algorithm baseline implementation, which allocates the flags in MPB by using MPBT mode [2]. The micro-benchmark results in this section are mainly divided into two groups, MO and ASO.

5.1 Pure Overhead

Figure 6 shows that C-LUTB and CPB algorithms on the MO and ASO achieve more than 97% overhead reduction for 48 threads. Figure 6(a) shows the average overhead of the Master thread that includes time to wait for the slaves and to release them on the SCC. While Figure 6(b) shows the average overhead to notify the Master thread by the slaves and waiting time for the exit (release) signals. For the ASO, the average overhead of the CPB approach is about 26 times smaller than S-MSB, because there is no contention to notify the Master and to access a shared signal variable. In our experiments, C-LUTB is always the fastest barrier, making it the ideal candidate to perform barrier synchronization regardless of increasing number of cores. For more than two cores, C-LUTB shows significantly better ASO than others.

5.2 Impact of Memory Access Mode

To study the impact of memory access mode, we reimplemented the algorithms (e.g. S-MSB and CPB) that have access to local memory (MPB). Figure 6 shows results from this experiment. In this experiment, the UC mode reduces the overhead on MPB-based algorithms, also it significantly improves the CPB algorithm as shown in Figure 6(a). The CPB-UC approach shows worse results by more than 96% compared to its overhead in the previous experiments. Figure 6(b) shows that CPB-UC reduces the overhead by 19.9% approximately for 48 threads. In addition, there is a slight overhead difference between CPB-UC and C-LUTB; we also note that there is no impact on the overhead with the increasing number of cores

6. CONCLUSIONS AND FUTURE WORK

It is attractive to support OpenMP programming on the SCC. In this work, we present our ongoing efforts towards an efficient OpenMP implementation for SCC, which is based on a modified GCC 2.6 compiler and on a custom run-time library. We are in process of supporting full-OpenMP parallelism to increase programmer productivity, reducing the design/development costs and time to market for the future many-core systems. We discussed the issues and requirements to support the OpenMP fork-join execution model on the Intel's SCC. In that sense, we believe that significant improvements can be achieved with the mindful usage of relevant architecture features. Key to reducing runtime overheads is an efficient barrier implementation, because OpenMP relies heavily on barrier operations to control threads in parallel. Therefore, we have implemented several barrier algorithms to optimize the OpenMP run-time library by using the specific hardware features of the SCC architecture. As part of a quantitative performance evaluation, our experimental results highlight that we can obtain a significant reduction in overhead for barrier algorithms by using Chain LUT-Polarity busy-wait approach. The C-LUTB is the best barrier synchronization which allows 98.5% (MO in Pure Overhead) faster synchronization than S-MSB and approximately 25.6% (MO in Pure Overhead) faster than CPB-UC algorithm. Also, C-LUTB has low usage for the SCC features and it shows low power consumption. Consequently, these represent a substantial decrease in the cost of managing parallelism. However, we first developed the evaluation criteria with micro-benchmark for choosing barrier synchronization OpenMP schemes of many-core system. Currently, we are dealing with the toughest challenge to support OpenMP data sharing on SCC: making shared data from main memory visible to all threads in presence of several OS instances, each with its own virtual memory space. As future work, we intend to deal with the necessity of en-

suring a consistent view of shared memory in absence of dedicated hardware cache coherence support.

7. REFERENCES

[1] H. Al-Khalissi, R. Buchty, and M. Berekovic. Efficient barrier synchronization for openmp-like parallelism on the intel scc.

[2] H. Al-Khalissi, A. Marongiu, and M. Berekovic. Low-overhead barrier synchronization for openmp-like parallelism on the single-chip cloud computer. http://darwin.bth.rwth-aachen.de/opus3/volltexte/2012/4383/, 2012.

[3] N. S. Arenstorf and H. F. Jordan. Comparing barrier algorithms. *Parallel Computing*, 12(2):157–170, 1989.

[4] E. AyguadÃl', M. GonzÃąlez, J. Labarta, X. Martorell, N. Navarro, and J. Oliver. Nanoscompiler: A research platform for openmp extensions. In *in the First EuropeanWorkshop on OpenMP*, pages 27–31, 1999.

[5] A. Basumallik and R. Eigenmann. Towards automatic translation of openmp to mpi. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 189–198, New York, NY, USA, 2005. ACM.

[6] O. A. R. Board. Openmp application program interface v.3.0. http://www.openmp.org/mp-documents/spec30.pdf, 2008.

[7] C. Brunschen and M. Brorsson. Odinmp/ccp - a portable implementation of openmp for c. *Concurrency*, 12(12):1193–1203, 2000. QC 20100525.

[8] J. M. Bull. Measuring synchronisation and scheduling overheads in openmp. In *In Proceedings of First European Workshop on OpenMP*, pages 99–105, 1999.

[9] B. M. Chapman, L. Huang, E. Biscondi, E. Stotzer, A. Shrivastava, and A. Gatherer. Implementing openmp on a high performance embedded multicore mpsoc. In *IPDPS*, pages 1–8. IEEE, 2009.

[10] I. Corporation. *SCC External Architecture Specification (EAS) - Revision 1.1*. http://communities.intel.com/docs/DOC-5852, Nov. 2010.

[11] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. Logp: towards a realistic model of parallel computation. *SIGPLAN Not.*, 28(7):1–12, July 1993.

[12] T. H. Dieter an Mey and W. Koschel. Pushing loop-level parallelization to the limit. Rome, Italy, 2002. Fourth European Workshop on OpenMP (EWOMP 2002).

[13] A. J. Dorta, P. LÃşpez, and F. de Sande. Basic skeletons in llc. *Parallel Computing*, 32(7-8):491–506, 2006.

[14] D. Eachempati, L. Huang, and B. M. Chapman. Strategies and implementation for translating openmp code for clusters. In *HPCC*, pages 420–431, 2007.

[15] Gomp. An openmp implementation for gcc. http://www.openmp.org/mp-documents/spec30.pdf.

[16] V. Gramoli, R. Guerraoui, and V. Trigonakis. Tm2c: a software transactional memory for many-cores. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 351–364, New York, NY, USA, 2012. ACM.

[17] L. Huang, B. M. Chapman, and Z. Liu. Towards a more efficient implementation of openmp for clusters via translation to global arrays. *Parallel Computing*, 31(10-12):1114–1139, 2005.

[18] W.-C. Jeun and S. Ha. Effective openmp implementation and translation for multiprocessor system-on-chip without using os. In *Proceedings of the 2007 Asia and South Pacific Design Automation Conference*, ASP-DAC '07, pages 44–49, Washington, DC, USA, 2007. IEEE Computer Society.

[19] J. Kim, S. Seo, and J. Lee. An efficient software shared virtual memory for the single-chip cloud computer. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 4:1–4:5, New York, NY, USA, 2011. ACM.

[20] F. Liu and V. Chaudhary. Extending openmp for heterogeneous chip multiprocessors. In *32nd International Conference on Parallel Processing (ICPP 2003), 6-9 October 2003, Kaohsiung, Taiwan*, page 161. IEEE Computer Society, 2003.

[21] A. Marongiu, P. Burgio, and L. Benini. Evaluating openmp support costs on mpsocs. pages 191–198, 2010.

[22] A. Marongiu, P. Burgio, and L. Benini. Supporting openmp on a multi-cluster embedded mpsoc. *Microprocess. Microsyst.*, 35(8):668–682, Nov. 2011.

[23] D. Millot, A. Muller, C. Parrot, and F. Silber-Chaussumier. From openmp to mpi: first experiments of the step source-to-source transformation tool. In *PARCO*, pages 669–676, 2009.

[24] K. O'Brien, K. M. O'Brien, Z. Sura, T. Chen, and T. Zhang. Supporting openmp on cell. *International Journal of Parallel Programming*, 36(3):289–311, 2008.

[25] G. Paoloni. How to benchmark code execution times on intel ia-32 and ia-64 instruction set architectures. 2010.

[26] J. Sartori and R. Kumar. Low-overhead, high-speed multi-core barrier synchronization. In *Proceedings of the 5th international conference on High Performance Embedded Architectures and Compilers*, HiPEAC'10, pages 18–34, Berlin, Heidelberg, 2010. Springer-Verlag.

[27] M. Sato, M. S. Shigehisa, K. Kusano, and Y. Tanaka. Design of openmp compiler for an smp cluster. In *In EWOMP âĂŹ99*, pages 32–39, 1999.

[28] S. Satoh, K. Kusano, and M. Sato. Compiler optimization techniques for openmp programs. In *Scientific Programming*, pages 9–2, 2001.

[29] K. Sivaramakrishnan, L. Ziarek, and S. Jagannathan. A Coherent and Managed Runtime for ML on the SCC. In *Proceedings of the Many-core Applications Research Community (MARC) Symposium at RWTH Aachen University*, pages 20–25, Nov. 2012.

[30] R. F. van der Wijngaart, T. G. Mattson, and W. Haas. Light-weight communications on intel's single-chip cloud computer processor. *SIGOPS Oper. Syst. Rev.*, 45(1):73–83, Feb. 2011.

[31] J. Wang, C. Hu, J. Zhang, and J. Li. Openmp compiler for distributed memory architectures. *SCIENCE CHINA Information Sciences*, 53(5):932–944, 2010.